PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46

A2

(11) International Publication Number: **WO 98/54643**

(43) International Publication Date: 3 December 1998 (03.12.98)

(21) International Application Number: PCT/US98/09138

(22) International Filing Date: 5 May 1998 (05.05.98)

(30) Priority Data:
08/866,419 30 May 1997 (30.05.97) US

(71) Applicant: SONY ELECTRONICS, INC. [US/US]; 1 Sony Drive, Park Ridge, NJ 07656–8003 (US).

(72) Inventors: OZCELIK, Taner; 542 Military Way, Palo Alto, CA 94306 (US). GADRE, Shirish, C.; 1265 N. Capitol Avenue #78, San Jose, CA 94132 (US).

(74) Agents: HUMPHREY, Thomas, W. et al.; Wood, Herron & Evans, L.L.P., 2700 Carew Tower, Cincinnati, OH 45202 (US).

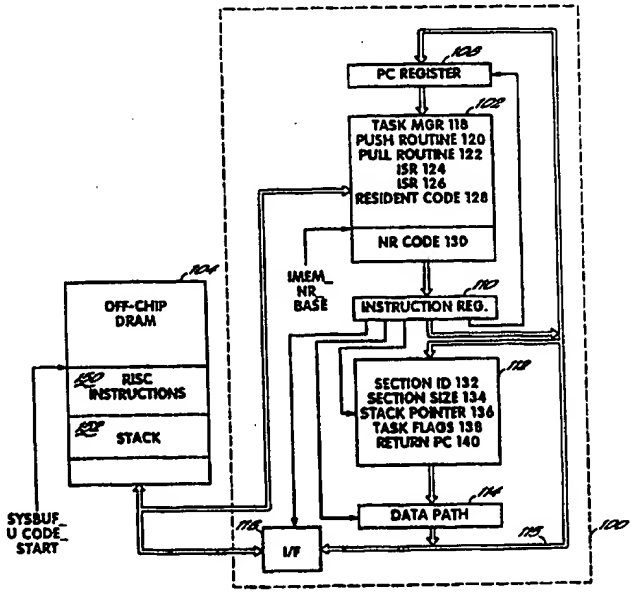
(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
Without international search report and to be republished upon receipt of that report.

(54) Title: TASK AND STACK MANAGER FOR DIGITAL VIDEO DECODING

(57) Abstract

A reduced instruction set CPU (100) is programmed to provide software–controlled task management, a stack (152), and to manage virtual instruction memory (150). The CPU performs a task management procedure (118) in which the CPU repeatedly checks (168) task flags, and if a task flag is set, performs the task associated with the set task flag. If multiple task flags are set, the highest priority task of those associated with set task flags is performed. Whenever a subroutine call is needed, the subroutine call is implemented by calling (172) a stack management routine. The stack management routine retrieves and stores (216) a return address into a location in DRAM (104) identified by a stack pointer, increments (218) the stack pointer, and then executes (234) a CALL instruction, causing program execution to sequence to the desired subroutine. At the end of each subroutine, a RETURN instruction is executed, in response to which, program execution returns to the stack management routine, and the stack management routine decrements (238) the stack pointer, loads (240) the previously–stored return address from a location in DRAM identified by the stack pointer register, and then causes (250) program execution to sequence to the loaded return address. The stack management routine also provides virtual instruction memory management, by determining (220, 244) whether a routine is resident in the on–chip instruction memory (102) available to the RISC CPU prior to calling or returning to the routine. If not, the virtual instruction memory management routine transfers (226, 248) the desired routine from off–chip DRAM (104) into the on–chip instruction memory, and then executes the call or return.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

TASK AND STACK MANAGER FOR DIGITAL VIDEO DECODING

Field of the Invention

The present invention relates to methods for controlling a reduced instruction set processor for digital video processing.

Background of the Invention

Techniques for digital transmission of video promise increased flexibility, higher resolution, and better fidelity. Recent industry collaborations have brought digital video closer to reality; digital video transmission and storage standards have been generated, and consumer digital video products have begun to appear. The move toward digital video has been encouraged by the commercialization of digital technologies in general, such as personal computers and compact discs, both of which have increased consumer awareness of the possibilities of digital technology.

Personal computers, which have recently become common and inexpensive, contain much of the computing hardware needed to produce digital video, including a microprocessor/coprocessor for performing numeric calculations, input and output connections, and a large digital memory for storing and manipulating image data. Unfortunately, personal computers are not suitable for consumer digital video reception, because the microprocessor in a personal computer is a general purpose processor, and typically cannot

perform the calculations needed for digital video fast enough to produce full-motion, high definition video output.

Accordingly, special purpose processors, particularly suited for performing digital video-related calculations, have been developed for use in digital video receivers for consumer applications. A specific processor of this kind is disclosed in commonly-assigned, copending U.S. Patent Application Serial No. 08/865,749, entitled SPECIAL PURPOSE PROCESSOR FOR DIGITAL AUDIO/VIDEO DECODING, filed by Moshe Bublil et al. on May 30, 1997, which is hereby incorporated by reference herein in its entirety, and a memory controller for use therewith is disclosed in commonly-assigned, copending U.S. Patent Application Serial No. 08/846,590, entitled "MEMORY ADDRESS GENERATION FOR DIGITAL VIDEO", filed by Edward J. Paluch on April 30, 1997, which is hereby incorporated herein in its entirety.

The above-referenced U.S. patent applications describe an application specific integrated circuit (ASIC) for performing digital video processing, which is controlled by a reduced instruction set CPU (RISC CPU). The RISC CPU controls computations and operations of other parts of the ASIC to provide digital video reception. As is typical of CPU's of many varieties, the CPU described in the above-referenced U.S. patent applications supports flow control instructions such as BRANCH, CALL and RETURN, as well as providing hardware interrupt services.

However, as is typical of RISC CPU design, for simplicity and savings of chip area, the functionality of the RISC CPU described in the above noted U.S. patent applications is limited in several ways. First, the RISC CPU does not include any support for a stack, and thus cannot support nested procedures calls, or nested interrupts. Whenever a CALL instruction is executed, the return address for that call is stored in a single on-chip register. If another CALL instruction is executed, the previous return address is lost,

and an error will result. Whenever an interrupt occurs, the RISC CPU branches to the associated interrupt service routine, and ignores all other interrupts until a RETURN is encountered at the end of the interrupt service routine.

A second limitation of the RISC CPU described in the above-referenced U.S. patent applications, is the size of the instruction memory available to the CPU. As described in the applications, the instruction memory is limited to 4096 instructions, which is a relatively small space; however, in some lower-density applications, the instruction memory might be further limited, for example to 2048 or fewer instructions, in order to conserve chip area.

These limitations of the RISC CPU described in the above-referenced U.S. patent applications, present difficulties in programming the RISC CPU for digital video decoding. Firstly, it is preferable to write software for a CPU in a modular fashion, such that those portions of the software that are used repeatedly, are incorporated into subroutines that can be called from other locations in the program. This approach reduces the total number of instructions in the program by making repeated use of common sets of instructions. Unfortunately, to take full advantage of this approach, it is often necessary to nest procedure calls, i.e., to permit subroutines to call other subroutines. If procedure calls cannot be nested, as is the case in the RISC CPU described in the above-referenced U.S. patent applications, it can be difficult to write software in an efficient, modular fashion.

Furthermore, it can be difficult to provide sufficient digital audio/video decoding functionality in a program which is limited to 2048 or even 4096 instructions. Thus, the small size of the instruction memory provided by the RISC CPU of the above-referenced U.S. patent applications presents a serious limitation on the functionality that can be provided.

Finally, difficulties arise from the nature of digital video processing. Digital video decoding involves a wide variety of digital processing tasks, some of which are time-critical (such as video "slice" decoding for display) and some of which are substantially less time critical (such as on-screen display and subtitle processing). As to time-critical tasks, the RISC CPU must detect when processing is needed and instruct other elements of the ASIC to respond appropriately, and must do so in a timely fashion.

One approach to providing this functionality, would be to associate each decoding task with an interrupt; when certain processing is needed, the associated interrupt is delivered to the CPU, causing the CPU to respond by branching to the associated interrupt service routine, and thereby performing the necessary actions. Unfortunately, when used in the RISC CPU described in the above-referenced U.S. patent applications, this approach could fail to provide the required responsiveness to time-critical tasks, because that RISC CPU does not permit nesting of interrupts. As one example of the problems that could occur, an interrupt for a non-time-critical task, might be delivered to the CPU just prior to an interrupt for a time-critical task. If this occurs, and if the interrupt service routine of the non-time-critical task is relatively lengthy, servicing of the time-critical task might be unacceptably delayed, since during the interrupt service routine for the non-time-critical task, no further interrupts will be serviced.

Summary of the Invention

In accordance with principles of the present invention, these difficulties are overcome by a novel method for controlling the RISC CPU which provides task management -- enabling rapid response to time-critical interrupts; and provides for stack management -- enabling nested procedure

calls allowing fully modular software structures; and provides virtual instruction memory management -- to provide an effectively unlimited virtual instruction space.

Specifically, in accordance with a first aspect, the invention provides a method of controlling a RISC CPU to provide management of tasks to be performed by the CPU. Tasks to be handled by the CPU are identified by binary task flags in a register of the CPU. The CPU performs a task management procedure in which the CPU repeatedly checks these task flags, and if a task flag is set, performs the task associated with the set task flag. If multiple task flags are set, the highest priority task of those associated with set task flags is performed.

In specific embodiments of this aspect of the invention, the task manager searches the flags in accordance with a recursive search procedure. Specifically, a group of flags is simultaneously checked to determine if any are set. If none or set, subsequent groups of flags are checked in a similar manner. If any flags are determined to be set, smaller subgroups of the group of flags are simultaneously checked to determine if any are set. This process repeats, checking increasingly smaller subgroups of flags, until finally, the location of the set flag is identified.

Tasks are scheduled in response to interrupts delivered to the RISC CPU. Specifically, when a processing task needs to be performed, an interrupt is delivered to the RISC CPU. In response to the interrupt, the CPU performs an interrupt service routine, and in this routine, one or more of the task flags may be set. Thereafter, upon conclusion of the interrupt, the set task flag will cause the task management procedure to perform the associated task.

The most time critical tasks can be provided special priority, by incorporating the necessary processing steps for the task, into an interrupt service routine. If this is done, once this interrupt service routine is initiated in

response to the associated interrupt, processing of the task will commence immediately, and will not be stalled.

In a second aspect, the present invention provides a method of controlling a RISC CPU to provide stack management, using an allocated area of off-chip dynamic random access memory (DRAM). Specifically, whenever a subroutine call is needed, the subroutine call is implemented by storing the address of the subroutine in a temporary register, and then executing a CALL instruction causing program execution to sequence to a stack management routine. The stack management routine retrieves the return address associated with the CALL instruction from a register of the RISC CPU, stores this return address into a location in DRAM identified by a stack pointer register in the RISC CPU, increments the stack pointer, and then executes a CALL instruction, causing program execution to sequence to the desired subroutine. At the end of each subroutine, a RETURN instruction is executed, which as a result of the foregoing, causes program execution to return to the stack management routine at the instruction following the previously-executed CALL instruction. After this RETURN, the stack management routine decrements the stack pointer, loads the previously-stored return address from a location in DRAM identified by the stack pointer register, and then executes a BRANCH instruction, causing program execution to sequence to the loaded return address. The use of a stack management routine in this manner, permits any number of subroutine calls to be nested within each other; in response to nested subroutine calls, return addresses are accumulated in the DRAM stack, for later retrieval in the appropriate order.

In a third aspect, the present invention provides a method of controlling a RISC CPU to provide virtual instruction memory management, using an allocated area of off-chip DRAM. Specifically, whenever a subroutine call is needed, the subroutine call is implemented in the above-

described manner, by storing the address of the subroutine in a temporary register, and then executing a CALL instruction causing program execution to sequence to a virtual instruction memory management routine, which can be part of the stack management routine described above, or could be implemented separately if the CPU provides hardware stack management. The virtual instruction memory management routine determines whether the desired subroutine is resident in the on-chip instruction memory available to the RISC CPU. If not, the virtual instruction memory management routine transfers the desired subroutine from off-chip DRAM into the on-chip instruction memory. Thereafter, the virtual instruction memory management routine executes a CALL instruction, causing program execution to sequence to the desired subroutine.

In the specific disclosed embodiment of this aspect of the present invention, as part of transferring the desired subroutine from off-chip DRAM into the on-chip instruction memory, the RISC CPU determines the number of instructions in the subroutine, and transfers only this number of instructions from off-chip DRAM into the on-chip instruction memory.

The virtual instruction memory management routine also, where necessary, computes a physical memory address of the desired subroutine, and causes program execution to sequence to the computed physical memory address. In this specific embodiment, some of the subroutines of the software for the CPU are permanently stored in the instruction memory, e.g., interrupt service routines and time-critical processing subroutines. At any given time, one of the remaining non-permanent subroutines is stored in the instruction memory, in a "non-resident" area of the instruction memory. As a result, the physical memory address of the first instruction of a non-permanent subroutine is always equal to the first address in this non-resident area of the instruction memory.

To keep track of which subroutine is stored in the non-resident area of the instruction memory, when a subroutine is swapped into instruction memory, the virtual memory address of the first instruction of the subroutine is stored in a register of the CPU. If a non-permanent subroutine is called, the virtual address of the called subroutine is compared to the virtual instruction memory address stored by the CPU. If the two are equal, this indicates that the desired subroutine is already resident in the instruction memory, and the step of swapping the subroutine into instruction memory is skipped, thus increasing the performance and reducing latency, both of which are critical in real-time control applications.

The above and other aspects, objects and advantages of the present invention shall be made apparent from the accompanying drawings and the detailed description thereof.

Brief Description of the Drawing

The accompanying drawings, which are incorporated in and constitute a part of this specification, illustrate embodiments of the invention and, together with the general description of the invention given above, and the detailed description of the embodiments given below, serve to explain the principles of the invention.

Fig. 1 is a block diagram of specific components of the application specific integrated circuit (ASIC) described in the above-referenced U.S. patent applications, showing in particular the off-chip dynamic random access memory (DRAM) and the data stored therein, the reduced instruction set central processing unit (RISC CPU) for controlling the ASIC, and components of the RISC CPU and the data stored therein;

Fig. 2A is a data structure diagram of one entry in the DRAM stack illustrated in Fig. 1, showing the data stored therein;

Fig. 2B is a data structure diagram of the task flags stored in one of the registers of the RISC CPU illustrated in Fig. 1, showing the data stored therein;

Fig. 3A is a flow chart of a typical interrupt service routine included in the permanent storage area of the instruction memory of the RISC CPU illustrated in Fig. 1;

Fig. 3B is a flow chart of the task manager included in the permanent storage area of the instruction memory of the RISC CPU illustrated in Fig. 1;

Fig. 3C is a flow chart of a typical task handler routine included in the permanent storage area of the instruction memory of the RISC CPU illustrated in Fig. 1;

Fig. 3D is a flow chart of the task flag search procedure used by the task manager illustrated in Fig. 3B;

Fig. 4A is a flow chart of the virtual instruction memory and stack management routine included in the permanent storage area of the instruction memory of the RISC CPU illustrated in Fig. 1, and particularly the PUSH routine portion thereof;

Fig. 4B is a flow chart of the virtual instruction memory and stack management routine included in the permanent storage area of the instruction memory of the RISC CPU illustrated in Fig. 1, and particularly the PULL routine portion thereof.

Detailed Description of Specific Embodiments

Referring to Fig. 1, the general structure of an ASIC, including a RISC CPU 100 for controlling digital audio/video decoding by the ASIC, can be discussed. The detailed structure of the RISC CPU 100 and of elements of the ASIC is described in the above-referenced U.S. patent applications. For

present purposes, only a few specifically relevant components of the RISC CPU of the ASIC will be discussed.

Specifically, the RISC CPU 100 includes an instruction memory 102 for storing instructions to be executed by the RISC CPU 100 in order to control digital audio/video processing performed by the ASIC. These instructions are initially loaded into instruction memory 102 from an off-chip dynamic random access memory (DRAM) 104, via a data bus 106 connected between DRAM 104 and the ASIC.

The address of an instruction in instruction memory 102 to be executed by the RISC CPU, is identified by a program counter (PC) register 108. The instruction read from instruction memory 102 is delivered to an instruction register 110, and used to control operations of the RISC CPU. As described in the above-referenced U.S. patent applications, instructions are 16-bits wide, and control a wide variety of operations of the RISC CPU, including operations of a register file 112, a data path 114, a memory interface 116, and other elements described in the above-referenced U.S. patent applications.

The program of instructions found in instruction memory 102 includes various routines, which are generally illustrated in Fig. 1. Specifically, a task manager routine 118 residing in instruction memory 102 (and discussed in further detail below with reference to Fig. 3B), is repeatedly executed during operation of the ASIC. Task manager routine 118 interacts with flags in a register of register file 112, to schedule and execute any of a number of tasks to perform needed audio/video processing.

Also included in instruction memory 102, is a stack and virtual instruction memory management routine, consisting of a PUSH routine 120 (further described below with reference to Fig. 4A), and a PULL routine 122 (further described below with reference to Fig. 4B). These routines are executed upon any subroutine call performed by the task handler, any specific

task or any interrupt service routine, and interact with an allocated region of DRAM 104 to provide virtual instruction memory management and permit nesting of subroutine calls.

Instruction memory 102 further includes several interrupt service routines such as 124 and 126. As noted earlier, these routines are activated by hardware of the RISC CPU upon receipt of an interrupt. As illustrated in further detail with reference to Fig. 3A, the interrupt service routine performs necessary audio/video processing control, and/or may set a task flag to cause an associated task to be scheduled by the task manager routine 118.

Instruction memory 102 also includes other resident routines such as 128, including routines for carrying out tasks scheduled by task manager 118, or subroutines called by tasks or interrupt service routines as part of processing.

All of the foregoing routines are permanently resident in instruction memory 102, that is, these routines are transferred into instruction memory 102 when the ASIC is initially booted, and are thereafter left unchanged. There is, however, an area of instruction memory 102 which may contain one of several "non-resident" routines, which are swapped into instruction memory 102 as needed during execution of a program by the RISC CPU. Specifically, at any given time, one of several "non-resident" routines 130 is stored in an allocated "non-resident" area of instruction memory 102. The "non-resident" area of instruction memory 102 is that area of the instruction memory located at addresses greater than the fixed address IMEM_NR_BASE. Instructions are loaded into the "non-resident" area of instruction memory 102 when those instructions are required for execution of a task or interrupt service routine, on an as-needed basis. The swapping is

controlled through execution of the PUSH and PULL routines 120 and 122, as described below in detail with reference to Figs. 4A and 4B.

Among the elements of the RISC CPU which are controlled by instructions, is a register file 112. Register file 112 includes a number, e.g., sixty-four, 32-bit registers which are used by RISC CPU 100 in performing calculations for audio/video digital signal decoding and processing. A few specific registers used by RISC CPU 100 as part of virtual instruction memory and stack management, are illustrated in Fig. 1 and will be discussed below.

A first register in register file 112 is the SectionID register 132, which stores data identifying which of several "non-resident" routines is stored in area 130 of instruction memory 102. This register is used in virtual memory management to determine whether a routine must be swapped into instruction memory 108, as discussed below with reference to Figs. 4A and 4B.

A second register in register file 112 is the SectionSize register 134, which stores data identifying the size of the current nonresident routine stored in area 130 of instruction memory 102. This register is used in virtual memory management to swap the appropriate number of instructions from DRAM 104 to instruction memory 102, and no more, thus minimizing the latency caused by swapping.

A third register in register file 112 is the StackPointer register 136, which identifies a location in the stack in DRAM 104 to which data should be written or from which data should be read as part of the stack management PUSH and PULL routines 120 and 122, discussed below with reference to Figs. 4A and 4B.

A fourth register in register file 112 is the TaskFlags register 138, which stores the flags used in scheduling tasks for execution as part of the task management routine 118, discussed below with reference to Fig. 3B.

A fifth register in register file 112 is the ReturnPC register 140, which stores a return address when a CALL instruction is executed by RISC CPU 100. Whenever a CALL instruction is executed by RISC CPU 100, the RISC CPU automatically stores the address of the instruction in instruction memory 102 immediately following the CALL instruction, into ReturnPC register 140. The address stored in ReturnPC register 140 is manipulated as part of the stack management PUSH and PULL routines 120 and 122, discussed below with reference to Figs. 4A and 4B, in order to provide a stack enabling nested subroutine calls.

Data path 114 in RISC CPU 100 responds to instructions generated from an instruction in instruction register 110, by performing arithmetic or logical operations on the contents of registers delivered from register file 112, and delivering the results of these computations to a data bus 115. The specific arithmetic or logical operation performed by data path 114 is determined by the opcode portion of the instruction in instruction register 110, as is described in substantially greater detail in the above-referenced U.S. patent applications.

The output of data path 114, which is schematically identified in Fig. 1 as a bus 115, is connected to an input of register file 112, allowing the results of arithmetic or logical computations performed by data path 114 to be stored into registers of register file 112. Furthermore, bus 115 is connected to memory interface 116, permitting the results of logical operations performed by data path 114 to be stored into DRAM memory 104, and further permitting data stored in DRAM memory 104 to be retrieved to bus 115 and stored, for example, into a register of register file 112.

The output of data path 114 can also be delivered to PC register 108, permitting a CALL or BRANCH instruction to directly load an address of an instruction in instruction memory 102, into PC register 108, thus causing

program execution to sequence to the new address. For this purpose, there is also a connection from instruction register 110 to bus 115, permitting immediate values in an instruction to be delivered to PC register 108 as part of a CALL or BRANCH instruction, and permitting immediate values in an instruction to be delivered to a register of register file 112.

The foregoing hardware structure is described in schematic fashion, for the purpose of illustrating the operation of the routines described in the following figures. The above-referenced U.S. patent applications describe these hardware structures in substantially greater detail, and in particular identify the instructions and specific paths of data flow provided by the RISC CPU.

As noted above, instructions are loaded into instruction memory 102 from DRAM 104 via bus 106. The instructions are stored in DRAM 104 in an allocated region 150 of DRAM memory 104. This allocated region, which stores for example at least 4096 32-bit instructions, begins at a base DRAM memory address SYSBUF_UCODE_START. Each subroutine used by RISC CPU 100 begins at a virtual instruction memory address, which is the location of the first instruction of the subroutine in the DRAM region 150, relative to the SYSBUF_UCODE_START address. Those subroutines with virtual instruction memory addresses which are less than IMEM_NR_BASE, are permanently stored in instruction memory 102. Subroutines with virtual instruction memory addresses greater than IMEM_NR_BASE are swapped into the nonresident portion 130 of instruction memory 102 on an as-needed basis.

When the ASIC is first booted, those instructions in DRAM 104 with virtual instruction memory addresses less than IMEM_NR_BASE, are loaded from region 150 of DRAM 104 into the permanent area of instruction memory 102. Thereafter, as part of execution of the virtual

instruction memory management PUSH and PULL routines 120 and 122, additional "non-resident" subroutines are loaded from region 150 of DRAM 104 into the non-resident area 130 of instruction memory 102, as discussed below with reference to Figs. 4A and 4B.

DRAM memory 104 also includes an allocated region 152 for storing a stack. As discussed in greater detail below, stack region 152 stores data necessary for performing nested subroutine calls, specifically, this region stores return addresses as well as data identifying non-resident subroutines needed to continue processing after a CALL instruction. The value in the StackPointer register 136 identifies a location in the stack region 152 of memory 104.

Referring now to Fig. 2A, the specific format of data stored in the stack region 152 of DRAM 104, can be described.

Each entry in the stack includes a 6-bit current SectionSize value, which identifies the number of instructions in any non-resident subroutine that was stored in the non-resident portion 130 of the instruction memory at the time that a CALL instruction was executed and the stack entry was stored in the stack by the PUSH routine 120. This 6-bit value is used by the PULL routine 122, if the PULL routine determines that it is necessary to swap the non-resident subroutine back into the non-resident portion 130 of the instruction memory to continue processing of instructions after the CALL.

Each stack entry also includes a 14-bit current SectionID value, which identifies the virtual instruction memory address of the non-resident subroutine that was stored in the non-resident portion 130 of the instruction memory at the time that a CALL instruction was executed and the stack entry was stored in the stack by the PUSH routine 120. This 14-bit value is used by the PULL routine 122, if the PULL routine determines that it is necessary to

swap the non-resident subroutine back into the non-resident portion 130 of the instruction memory to continue processing of instructions after the CALL.

Each entry in the stack further includes a single bit NR, which identifies whether the CALL instruction which caused the PUSH routine 120 to store an entry on the stack, was in a non-resident routine. The NR flag is used by the PULL routine to determine whether the instructions after the CALL are in a non-resident routine, and thus determine whether the non-resident routine that was in memory at the time the CALL instruction was executed, must be in memory to continue processing of instructions after the CALL.

Each entry in the stack finally includes an 11-bit value ReturnPC, which identifies the physical instruction memory address of the instruction immediately following the CALL instruction which caused the PUSH routine 120 to store an entry on the stack. The PULL routine 122 uses the ReturnPC value to cause program execution to sequence to the instruction immediately following the CALL instruction.

Referring now to Fig. 2B, the specific structure of the TaskFlags register 138 can be described. This register includes thirty-two single-bit flags. As noted above and elaborated below, to schedule execution of a task, one of the bits of this register are set, thereby causing the task manager 118 to execute the corresponding task. The bits in register 138 are in order of priority of the tasks; higher priority tasks are positioned in lower-order bits of the task register. Thus, for example, one of the lower-order bits might be associated with a parse_slice task, which is a relatively time-critical activity in digital audio/video decoding, involving the parsing of a "slice" of video. A higher-order bit might be associated with a parse_upper_layer_video task, which is a less time critical activity in digital audio/video decoding, involving the parsing of headers of packets of digital information. An even higher order

bit might be associated with a parse_subpicture task, which is a low priority task in digital audio/video decoding, involving parsing of information for generating subpicture video information.

Referring now to Fig. 3A, an interrupt service routine such as 124 or 126, includes a first step 160 in which the interrupt flag for the service routine is cleared, so that a subsequent interrupt of the same type can be delivered to the RISC CPU. Next, in step 162, any number of appropriate real-time processing steps may be performed. As noted above, for relatively high-priority real-time control tasks, various real-time processing steps may be incorporated into the interrupt service routine, rather than included in a task scheduled by the task manager. This may be necessary for those tasks that cannot tolerate latency, such as many low-level video decoding operations. After any steps 162, the interrupt service routine may include a step 164, in which one of the task handler flag bits is set in order to schedule a task for subsequent execution. As noted, lower priority tasks include relatively straightforward interrupt service routines, which simply set a task flag and return. Other routines may perform certain time critical operations in step 162, and then set a task flag in step 164 to schedule other, less time critical tasks. Still other routines might perform all operations in step 162, and not schedule any operations with the task manager in step 164. In any case, after steps 160, 162 and 164, a RETURN instruction at the end of the interrupt service routine terminates servicing of the interrupt.

Referring to Fig. 3B, the task manager 118 includes a number of steps for responding to the scheduling of tasks by previously-executed interrupt service routines. Specifically, the task manager executes a continuous loop, the first step 168 of which determines whether any of the task flags is set. If no flags are set, the task manager 118 will continually re-execute step 168 until a flag is set by an interrupt service routine. Once a flag

is set, the task manager 118 will proceed from step 168 to step 170, in which the task manager determines which task flag is set.

It will be appreciated that multiple task flags might be set at any time, such as is illustrated for example in Fig. 2B. In such a situation, task manager 118 services the tasks in accordance with a prioritization. As discussed above, the higher priority tasks are associated with lower-order bits of the task flag register. Accordingly, in step 170, task manager 118 scans the task bits in the task flag register, beginning with the least significant bit, attempting to locate the set task flag associated with the highest priority task. (Details of this operation are discussed below in connection with Fig. 3D.) Ultimately, the highest priority set bit is found, and the task manager responds to this bit by identifying the virtual instruction memory address of the first instruction of the associated task handler. This virtual address is stored in a temporary register. Next in step 172, the task manager executes a CALL instruction, causing execution to sequence to the stack and swap manager PUSH routine 120. As discussed in detail below, the stack and swap manager uses the virtual instruction memory address stored in the temporary register to commence execution of the desired task handler. After step 172, the task manager returns to step 168 to search for another set task flag.

Referring to Fig. 3C, a typical task handler 128 that is activated by task manger 118 can be discussed. A typical task handler includes a step 174 in which appropriate operations for carrying out the requested task are executed. This may include mathematical operations using the RISC CPU datapath, communications with functional units of the ASIC, or interaction with DRAM memory. In addition to such operations, a task handler may also include a step 176 in which the task handler calls a subroutine of instructions, for example to perform a common communication or memory management task. To call a subroutine, the task handler stores the virtual instruction

memory address of the first instruction of the desired subroutine into a temporary register, and then executes a CALL instruction, causing execution to sequence to the stack and swap manager PUSH routine 120. The PUSH routine then causes execution to sequence to the desired subroutine. After the subroutine call, a step 178, including additional operations, may be included in the task handler, for example, using data generated by the subroutine called in step 176. A task handler may include any number of such task-related instructions or subroutine calls, in any desired order. The PUSH and PULL routines 120 and 122 ensure that a virtually unlimited number of subroutine calls can be performed in a task handler without concern for nesting of subroutine calls, and without concern for whether called subroutines are in the permanently resident portion of the instruction memory 102 or alternatively must be swapped into the nonresident portion of instruction memory 102 to locations above the address IMEM_NR_BASE.

Referring now to Fig. 3D, the method used by the task manager to identify the highest priority task having a set flag can be detailed. As illustrated in Fig. 2B, there are 32 task flags, stored in a 32-bit register of the RISC CPU 100. These task flags are ordered, from the highest to lowest priority. Once task manager 118 determines that there is a set flag in this register (which is determined by comparing the entire register to zero using the BZ instruction provided by RISC CPU), task manager 118 follows a fast search procedure to locate the set task flag. This procedure evaluates the task flags in groups, to first locate the highest-priority group having a set task flag. Next, individual flags in the group are evaluated to identify the highest-priority set task flag in the group.

In a first step 180, a temporary register in RISC CPU 100 is initialized to a value of zero. The value in this register is updated to indicate the number of the task having the set task flag. Next, in step 182, the least

significant six bits in the task flag register are compared to the binary value "000000"; essentially, this determines whether any of the bits zero through five of the task flag register 138 (the six least significant task flags) are set. If none of the six least significant task flags are set, then in step 184, the value in the temporary register is incremented by six, indicating that the next group of bits tested will be bits six through eleven. Then, in step 186, the task flag bits are shifted six places to the right, thus positioning bits six through eleven of the task flag register 138 in the six least significant places. Processing then returns to step 182, in which bits six through eleven of the task flag register are compared to the value "000000".

This loop continues, until in step 182, a group of bits has one or more task flag set. At this point, processing proceeds from step 182 to step 190. In step 190, the task manager determines whether the least significant bit (LSB) of the task flags is equal to "0". If so, then the set task flag associated with the highest priority task has been found. However, if the LSB of the task flags is not "0", then in step 192, the value in the temporary register is incremented by one, indicating that the next bit is being tested. Then, in step 194, the task flag bits are shifted one place to the right. Finally, step 190 is repeated, to again test the LSB of the task flag bits to "0".

These steps repeat in this manner, until in step 190, a task flag bit has a "1" value. At this point, the set task flag associated with the highest priority task has been found, and the number of the bit is stored in the temporary register. Accordingly, in step 200, the contents of the temporary register is used as an index to lookup the virtual address of the associated task handler from a table permanently stored in instruction memory 102. In step 202, the virtual address obtained from the table, is then stored into the appropriate temporary register so that subsequent step 172 of the task manager will call the appropriate task handler.

Referring now to Fig. 4A, the virtual instruction memory and stack manager PUSH routine can be more fully described. This routine begins with a step 210, in which the value IMEM_NR_BASE, is compared to the return address found in the ReturnPC register 140. The return address in the ReturnPC register 140 contains the instruction memory address of the instruction after the CALL which activated the PUSH routine; if this address is greater than IMEM_NR_BASE, this indicates that the calling routine is in the nonresident area of the instruction memory, and processing proceeds to step 212 in which a flag "NR" is set to "1"; if the return address is less than IMEM_NR_BASE, then the calling routine is in the permanent area of the instruction memory, and processing proceeds to step 214, in which the NR flag is set to "0". The NR flag contains important information, used by the PULL routine. Specifically, the PULL must know whether a nonresident routine must be loaded into the instruction memory prior to returning execution to the return address; the NR flag supplies this information.

After step 212 or 214, processing proceeds to step 216, in which the current value in the ReturnPC register, the current value of the NR flag, the current value of the SectionID register, and the current value of the SectionSize register, are concatenated together in the format shown in Fig. 2A, and stored in DRAM 104 at the location identified by the StackPointer register. This step "pushes" the stack with the return address and related information, for later retrieval. Thereafter, in step 218, the value in the StackPointer register 136 is incremented, so that the stack is ready to receive data in the next "push".

After pushing data onto the stack, the PUSH routine prepares to call the desired subroutine identified by the calling routine. First, the PUSH routine must determine if the desired routine is in the permanent area of the instruction memory 102, or alternatively is a nonresident routine. To

determine this, in step 220, the PUSH routine compares the virtual instruction memory address stored in the temporary register by the calling routine, to the IMEM_NR_BASE value.

If the virtual address in the temporary register is greater than IMEM_NR_BASE, this indicates that the desired subroutine is a "nonresident" routine, and may not be present in the instruction memory. In such a situation, processing proceeds to step 222, in which the virtual address in the temporary register is compared to the current value in the SectionID register. Since the SectionID register always contains the virtual instruction memory address of the nonresident routine stored in the nonresident area of the instruction memory, if the virtual address of the desired subroutine is not equal to the SectionID register, this indicates that the desired subroutine is not currently in the nonresident area of instruction memory 102.

In this circumstance, processing proceeds to step 224, where the PUSH routine prepares to swap the desired subroutine into the nonresident area of the instruction memory 102. Specifically, the swap routine first determines the size of the desired subroutine, and stores this size into the SectionSize register 134. The size of each subroutine may be stored in a number of ways, such as in a table found in the permanent or nonresident portions of the RISC CPU program. In the specific embodiment discussed herein, however, the size of each subroutine is stored in the subroutine itself, at a fixed offset (e.g., eight instructions) from the first instruction of the subroutine. This approach permits individual subroutines to be reassembled separately from the remainder of the program. In this embodiment, therefore, step 224 comprises loading, from the DRAM location which is eight instructions into the desired subroutine, the number of instructions in the subroutine, and storing this number in the SectionSize register 134.

After step 224, in step 226, the actual swap of instructions to the instruction memory 102 is performed. Specifically, the RISC CPU delivers a command to the memory controller of the ASIC, to swap the number of 32-bit instructions identified by the SectionSize register, into the instruction memory 102 at locations starting at IMEM_NR_BASE, from DRAM 104, starting at the DRAM address which is equal to the virtual instruction memory address stored in the temporary register by the calling routine, incremented by the offset value SYSBUF_UCODE_START. In response to this command, as described in detail by the above-referenced U.S. patent applications, the memory controller of the ASIC delivers SectionSize instructions to instruction memory 102, which are stored therein.

After step 226, a new subroutine has been stored in the nonresident area of instruction memory 102. Accordingly, in step 228, the virtual instruction memory address stored in the temporary register by the calling routine, is stored in the SectionID register 132, thus indicating that this subroutine is now in the nonresident portion of the instruction memory 102.

After step 228, the PUSH routine proceeds to step 230, in which it computes the physical memory address of entry point into the desired subroutine in instruction memory 102, in preparation for executing the desired subroutine. If in step 222, it is determined that the desired nonresident subroutine is in the nonresident portion of the instruction memory, processing also proceeds directly to step 230.

In either case, in step 230, the desired subroutine is a nonresident subroutine. Therefore, the location of the subroutine is IMEM_NR_BASE, i.e., it is located at the beginning of the nonresident portion of the instruction memory 102. Accordingly, the physical instruction memory address for the entry point of the routine is generated using

IMEM_NR_BASE, rather than the virtual instruction memory address of the routine.

As discussed above, the calling routine stores the virtual instruction memory address of the desired subroutine into a temporary register before calling the PUSH routine 120. At the same time, the calling routine can identify an entry point into the desired subroutine which is other than the first instruction of the subroutine, by storing an offset address into the temporary register along with the virtual instruction memory address of the subroutine. Accordingly, in step 230, when computing the physical memory address of the entry point into the desired subroutine in instruction memory 102, the PUSH routine adds the offset value stored in the temporary register by the calling routine, to the value IMEM_NR_BASE, to generate the actual entry point into the desired subroutine.

A similar process is performed where the desired subroutine is located in the permanent portion of the instruction memory. Specifically, if at step 220, it is determined that the desired subroutine is in the permanent portion of the instruction memory, then in step 232, the entry point into the subroutine is computed by adding the offset value stored in the temporary register by the calling routine, to the virtual instruction memory address for the subroutine, to generate the entry point into the desired subroutine.

Referring now to Fig. 4B, after step 230 or 232, the physical instruction memory address of the entry point into the desired subroutine has been computed, and accordingly a CALL instruction is executed in step 234, causing the desired subroutine to begin executing at the desired entry point. As a result of the execution of this CALL instruction, the first instruction after the CALL instruction (i.e., the first instruction of the PULL routine) will be stored in the ReturnPC register 140. Thus, whenever any subroutine reaches

its end, and executes a RETURN instruction, processing returns to the first instruction of the PULL routine.

In the first step 238 of the PULL routine, the stack pointer is decremented, and then in step 240, previously stored values are loaded from the stack. Specifically, the ReturnPC, NR flag, SectionSize and SectionID values previously stored on the stack are retrieved.

At this point, the PULL routine must determine, first, whether the ReturnPC value identifies a nonresident routine, or a subroutine in the permanent area of instruction memory 102. In the former situation, the PULL routine must ensure that the nonresident calling routine is in the nonresident portion of instruction memory 102, before processing can proceed.

Accordingly, in step 242, the PULL routine checks the NR flag retrieved from the stack. If the NR flag has a value of "1", this indicates that the calling routine is a nonresident routine. In this situation, it may be necessary to swap the calling routine into the nonresident portion of the instruction memory 102, before processing can proceed.

Accordingly, if the NR flag is "1" in step 242, in step 244, the PULL routine compares the current value of the SectionID register 132, to the SectionID value retrieved from the stack in step 240, thus determining whether the nonresident routine currently in the nonresident portion of instruction memory 102, which is identified by the SectionID register 132, is the nonresident calling routine which is identified by the SectionID value retrieved from the stack. If the two are equal, then it is not necessary to swap the nonresident calling routine into the instruction memory 102, since the routine is already in the nonresident area of instruction memory 102.

If, however, the SectionID value retrieved from the stack is not equal to the SectionID value in register 132, then the nonresident calling routine must be swapped into the nonresident area of the instruction memory.

Accordingly, in this situation, the PULL routine proceeds from step 244 to step 246, in which the SectionID value retrieved from the stack is stored in the SectionID register 132, and the SectionSize value retrieved from the stack is stored in the SectionSize register 134. Then, in step 248, the instructions for the calling routine are swapped from DRAM into the nonresident portion of the instruction memory 102. Specifically, the RISC CPU delivers a command to the memory controller of the ASIC, to swap the number of 32-bit instructions identified by the SectionSize register, into the instruction memory 102 at locations starting at IMEM_NR_BASE, from DRAM 104, starting at the DRAM address which is equal to the virtual instruction memory address stored in the SectionID register, incremented by the offset value SYSBUF_UCODE_START. In response to this command, as described in detail by the above-referenced U.S. patent applications, the memory controller of the ASIC delivers SectionSize instructions to instruction memory 102, which are stored therein.

After step 248, the nonresident calling routine is stored in the nonresident portion of instruction memory 102. Accordingly, in step 250, the PULL routine

executes a BRANCH instruction, causing program execution to sequence to the instruction identified by the ReturnPC register.

As noted above, the above steps only occur if the calling routine is a nonresident routine (as determined at step 242) and the nonresident calling routine is not current stored in the nonresident portion of instruction memory 102 (as determined at step 244). If the calling routine is in the permanent area of the instruction memory 102 (as determined at step 242), or if a nonresident calling routine is already stored in the nonresident portion of instruction memory 102 (as determined at step 246), then processing proceeds directly to step 250, bypassing the swapping steps 246 and 248. This bypass

substantially reduces the latency associated with returning from called subroutines, in that nonresident routines are only swapped into the nonresident area of the instruction memory 102 when and if those routines are needed and are not already present in instruction memory 102.

While the present invention has been illustrated by a description of various embodiments and while these embodiments have been described in considerable detail, it is not the intention of the applicants to restrict or in any way limit the scope of the appended claims to such detail. Additional advantages and modifications will readily appear to those skilled in the art. For example, principles of the present invention may be applied to task and stack management in reduced instruction set central processing units used in applications other than digital video signal processing. Furthermore, principles of the present invention might be expanded, to permit multiple nonresident subroutines to be independently swapped into or out of a nonresident area of instruction memory 102. The invention in its broader aspects is therefore not limited to the specific details, representative apparatus and method, and illustrative example shown and described. Accordingly, departures may be made from such details without departing from the spirit or scope of applicant's general inventive concept.

What is claimed is:

Claims

1. A method of controlling a reduced instruction set central processing unit to provide management of a plurality of tasks to be performed by said central processing unit, comprising

providing a register of said central processing unit for storing a plurality of binary task flags, each of said task flags being associated with one of said plurality of tasks,

providing sequences of instructions in an instruction memory of said central processing unit, said sequences of instructions, when executed by said central processing unit, performing one of said plurality of tasks,

repeatedly executing a task management procedure in said central processing unit, said task management procedure comprising the steps of

determining whether any of said task flags is set, and if a flag is set

executing the sequence of instructions associated with a set task flag.

2. The method of claim 1, wherein said task management procedure determines whether any of said task flags is set, by the steps of

determining whether any of a group of said task flags is set, and if so, determining which specific task flag of said group is set.

3. The method of claim 2 wherein said task management procedure determines which specific task flag of a group is set by determining whether any of a first subgroup of task flags of said group is set, and if so, determining which specific task flag of said subgroup is set, otherwise, determining whether any of a second different subgroup of task flags of said

group is set, and if so, determining which specific task flag of said subgroup is set.

4. The method of claim 2 wherein said task management procedure performs said determining step on two or more different groups of said task flags.

5. The method of claim 1, wherein, if multiple task flags are set during said task management procedure, said task management procedure selects a highest priority task of those tasks associated with set task flags, and executes said highest priority task.

6. The method of claim 1, further comprising

responding to an interrupt signal delivered to said central processing unit, by executing an interrupt service routine, said interrupt service routine comprising the step of setting one of said task flags, to thereby cause subsequent execution of the sequence of instructions performing the associated task.

7. The method of claim 1, further comprising

responding to an interrupt signal delivered to said central processing unit indicating need for execution of a time-critical task, by executing an interrupt service routine, said interrupt service routine comprising instructions for performing said time-critical task.

8. A method of controlling a reduced instruction set central processing unit to sequence from a calling routine in an instruction memory of said central processing unit, to a desired subroutine in said instruction

memory, while providing management of a stack in an off-chip memory accessible to said reduced instruction set central processing unit, comprising

storing a stack pointer value identifying a location in said off-chip memory, into a stack pointer register of said central processing unit,

storing an address of said desired subroutine in a temporary register of said central processing unit,

executing a CALL instruction in said central processing unit, said CALL instruction causing said central processing unit to execute a stack management routine in said instruction memory, and to store a return address identifying the location of an instruction subsequent to said CALL instruction in said instruction memory, into a return address register of said central processing unit,

wherein execution of said stack management routine comprises

retrieving said return address from said return address register of said central processing unit,

storing said return address into a location in said off-chip memory identified by said stack pointer value,

incrementing said stack pointer value and storing the incremented stack pointer value into said stack pointer register, and then

retrieving said address of said desired subroutine from said temporary register of said central processing unit,

executing a CALL instruction causing said central processing unit to execute said desired subroutine.

9. The method of claim 8 wherein said desired subroutine address stored in said temporary register identifies a base subroutine address and an entry point offset, and stack management routine executes said desired

subroutine by executing a CALL instruction causing said central processing unit to execute an instruction at a memory address equal to said base subroutine address incremented by said entry point offset.

10. The method of claim 8 including steps for returning to said calling routine after execution of said desired subroutine, wherein

said desired subroutine ends in a RETURN instruction which causes said central processing unit to commence execution of instructions in said stack management routine at an instruction subsequent to said CALL instruction of said stack management routine, and

execution of said stack management routine subsequent to said CALL instruction further comprises:

decrementing said stack pointer value and storing the decremented stack pointer value into said stack pointer register,

loading the previously-stored return address from a location in said off-chip memory identified by said stack pointer value, and then

executing a BRANCH instruction causing said central processing unit to commence execution of instructions at an instruction identified by said return address.

11. A method of controlling a reduced instruction set central processing unit to sequence from a calling routine in an instruction memory of said central processing unit, to a desired subroutine in said instruction memory, while providing management of instructions stored in an off-chip memory accessible to said reduced instruction set central processing unit, comprising

storing an address of said desired subroutine in a temporary register of said central processing unit.

executing a CALL instruction in said central processing unit, said CALL instruction causing said central processing unit to execute a virtual instruction memory management routine in said instruction memory, and to store a return address identifying the location of an instruction subsequent to said CALL instruction in said instruction memory, into a return address register of said central processing unit,

wherein execution of said virtual instruction memory management routine comprises

determining whether said desired subroutine is resident in said instruction memory, and if not, loading said desired subroutine from said off-chip memory into said instruction memory, and then

executing a CALL instruction causing said central processing unit to proceed to a first instruction of said desired subroutine in said instruction memory.

12. The method of claim 11, wherein execution of said virtual instruction memory management routine comprises determining a number of instructions in said desired subroutine, and then loading only said number of instructions in said desired subroutine into said instruction memory.

13. The method of claim 11, further comprising

storing a stack pointer value identifying a location in said off-chip memory, into a stack pointer register of said central processing unit, wherein

execution of said virtual instruction memory management routine further comprises

retrieving said return address from said return address register of said central processing unit,

storing said return address into a location in said off-chip memory identified by said stack pointer value, and

incrementing said stack pointer value and storing the incremented stack pointer value into said stack pointer register.

14. The method of claim 11, further comprising

storing a stack pointer value identifying a location in said off-chip memory, into a stack pointer register of said central processing unit, wherein

execution of said virtual instruction memory management routine further comprises

storing an identification of a subroutine currently resident in said instruction memory into a location in said off-chip memory identified by said stack pointer value, and

incrementing said stack pointer value and storing the incremented stack pointer value into said stack pointer register.

15. The method of claim 11, further comprising steps for returning to said calling routine after execution of said desired subroutine, wherein

said desired subroutine ends in a RETURN instruction which causes said central processing unit to commence execution of instructions at an instruction subsequent to said CALL instruction of said virtual instruction memory management routine, and

execution of said virtual instruction memory management routine subsequent to said CALL instruction further comprises:

determining whether said calling routine is resident in said instruction memory, and if not, loading said calling routine from said off-chip memory into said instruction memory, and then

executing a BRANCH instruction causing said central processing unit to commence execution of instructions at an instruction identified by said return address.

16. The method of claim 13, wherein execution of said virtual instruction memory management routine subsequent to said CALL instruction further comprises

decrementing said stack pointer value and storing the decremented stack pointer value into said stack pointer register,

loading the previously-stored return address from a location in said off-chip memory identified by said stack pointer value, and then

executing a BRANCH instruction causing said central processing unit to commence execution of instructions at an instruction identified by said return address.

17. The method of claim 11, wherein

execution of said virtual instruction memory management routine prior to said CALL instruction further comprises storing an indication of whether said calling routine is permanently stored in said instruction memory, and

execution of said virtual instruction memory management routine subsequent to said CALL instruction comprises loading said indication of whether said calling routine is permanently stored in said instruction memory, and if said calling routine is permanently stored in said instruction memory, proceeding directly to executing said BRANCH instruction.

18. The method of claim 11, wherein execution of said virtual instruction memory management routine further comprises computing an address of a first instruction of said desired subroutine in said instruction memory, for use in said subsequent execution of said BRANCH instruction.

19. The method of claim 11, wherein execution of said virtual instruction memory management routine further comprises storing an identification of said desired subroutine into a current section register of said central processing unit, when said desired subroutine is loaded from said off-chip memory into said instruction memory.

20. The method of claim 19, further comprising steps for returning to said calling routine after execution of said desired subroutine, wherein

said desired subroutine ends in a RETURN instruction which causes said central processing unit to commence execution of instructions at an instruction subsequent to said CALL instruction of said virtual instruction memory management routine, and

execution of said virtual instruction memory management routine subsequent to said CALL instruction further comprises:

determining whether said calling routine is identified by said current section register of said central processing unit, and

if not, loading said calling routine from said off-chip memory into said instruction memory of said central processing unit and storing an identification of said calling routine into said current section register of said central processing unit, and then

executing a BRANCH instruction causing said central processing unit to commence execution of instructions at an instruction identified by said return address.

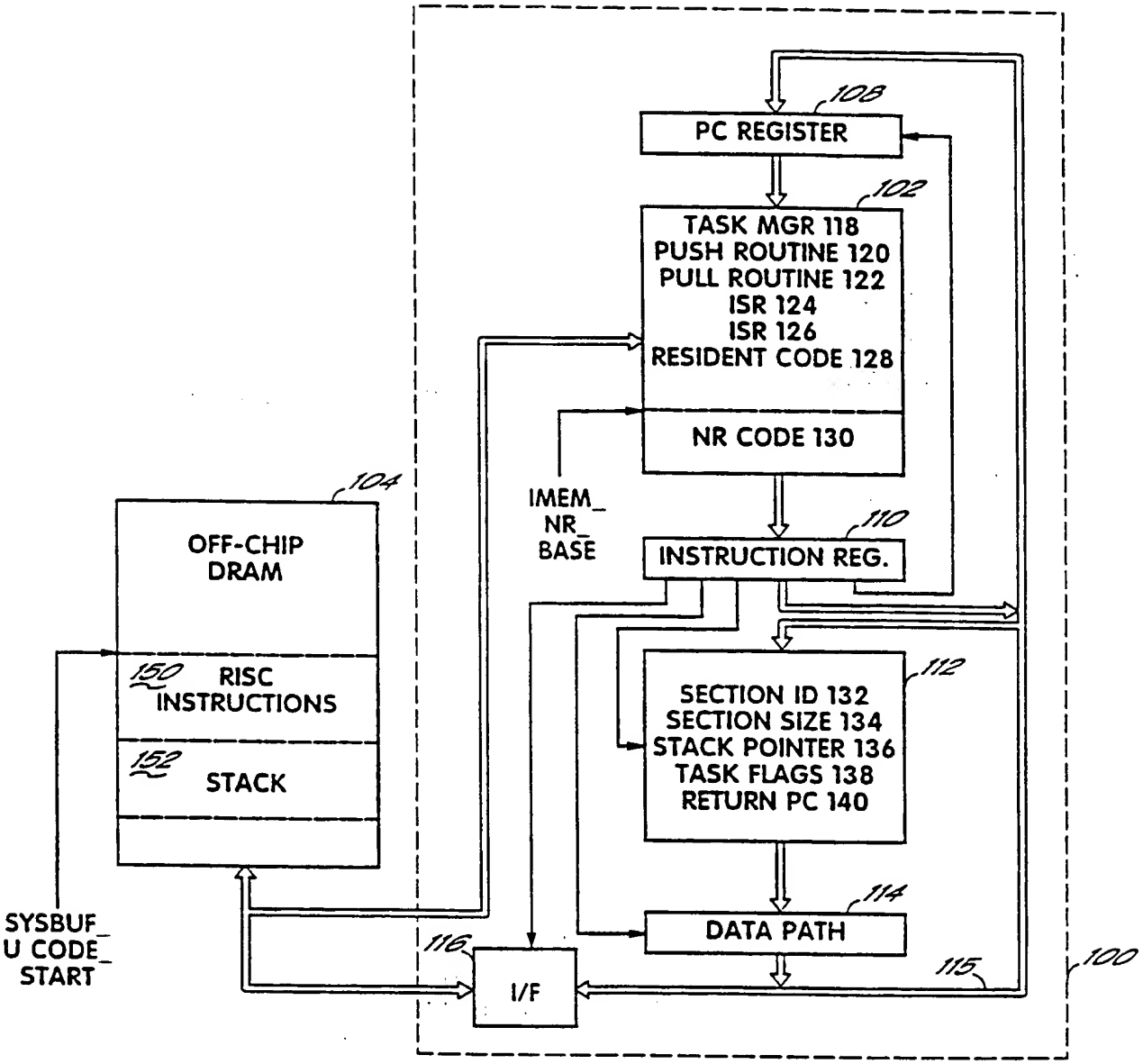


FIG. 1

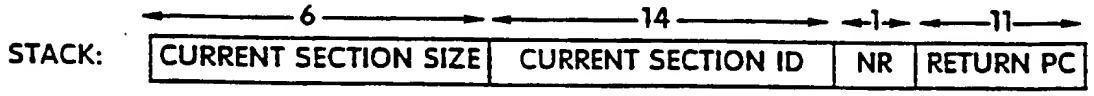


FIG. 2A

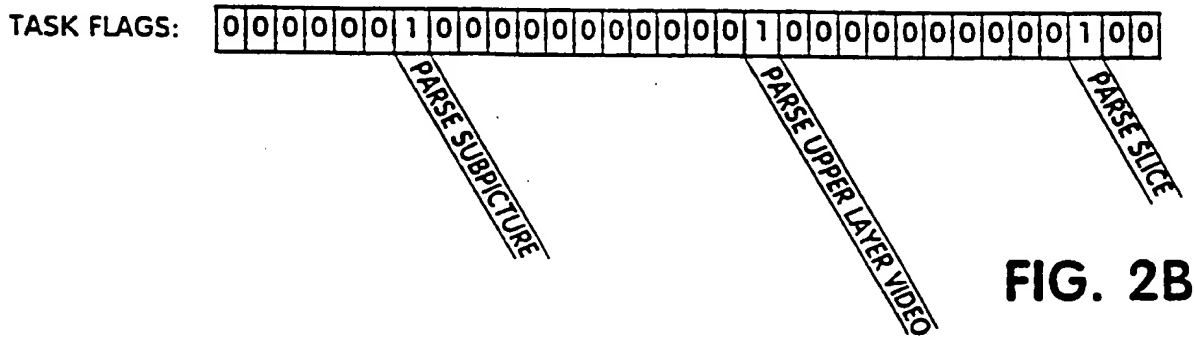


FIG. 2B

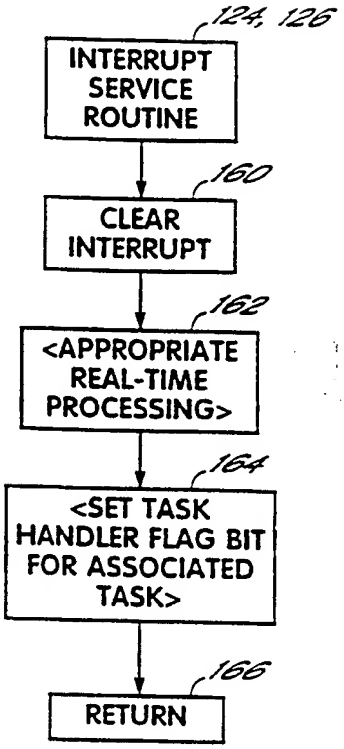


FIG. 3A

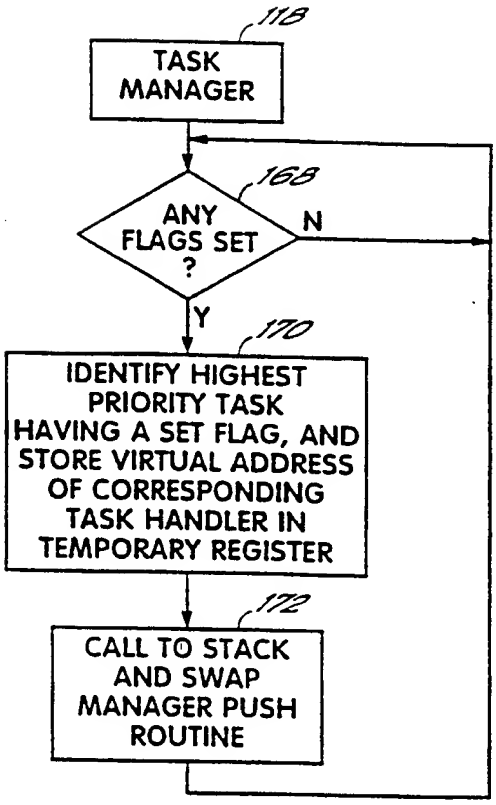


FIG. 3B

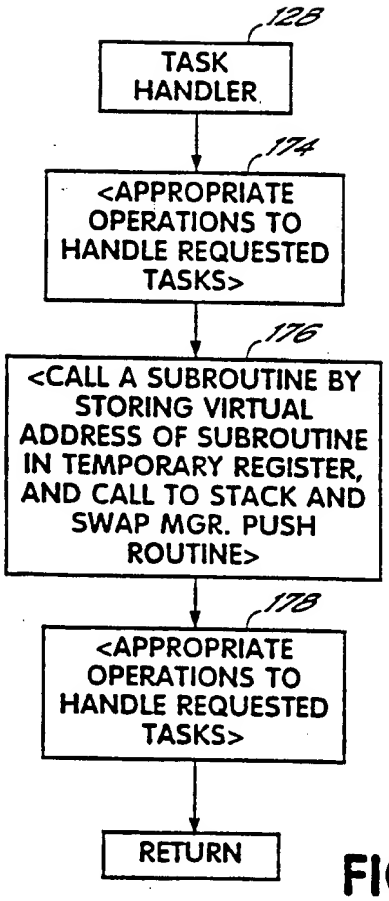


FIG. 3C

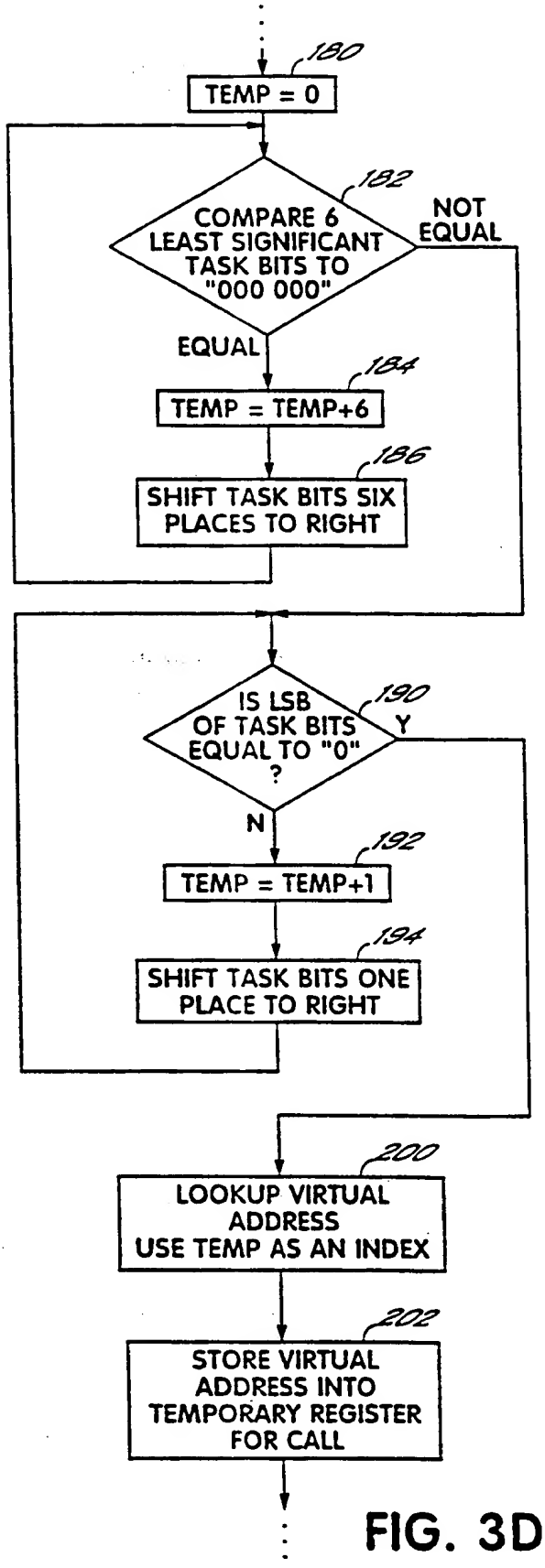


FIG. 3D

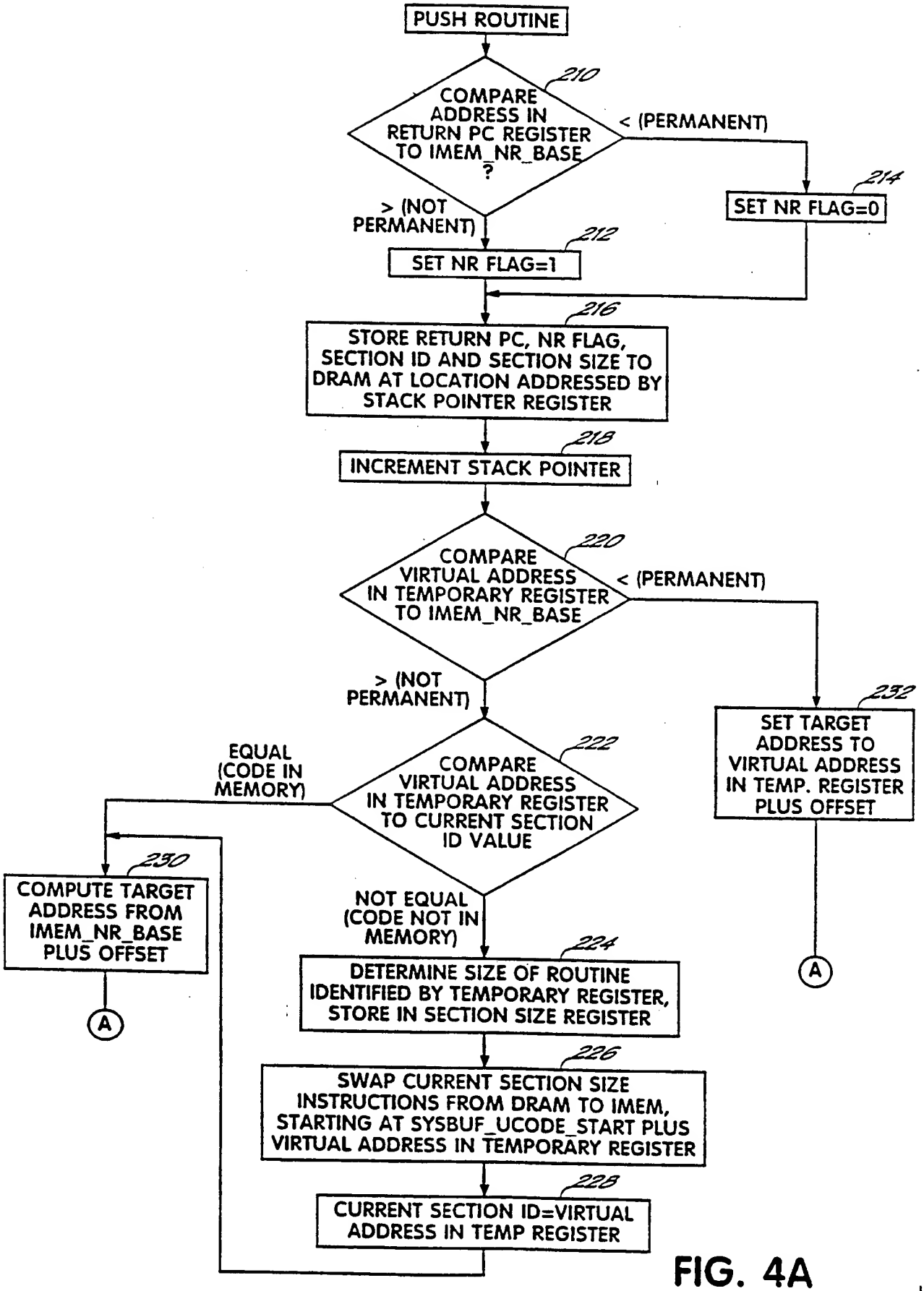

PUSH ROUTINE
210
COMPARE ADDRESS IN RETURN PC REGISTER TO IMEM_NR_BASE ?
< (PERMANENT)
214
SET NR FLAG=0
> (NOT PERMANENT)
212
SET NR FLAG=1
216
STORE RETURN PC, NR FLAG, SECTION ID AND SECTION SIZE TO DRAM AT LOCATION ADDRESSED BY STACK POINTER REGISTER
218
INCREMENT STACK POINTER
220
COMPARE VIRTUAL ADDRESS IN TEMPORARY REGISTER TO IMEM_NR_BASE
< (PERMANENT)
> (NOT PERMANENT)
232
SET TARGET ADDRESS TO VIRTUAL ADDRESS IN TEMP. REGISTER PLUS OFFSET
222
COMPARE VIRTUAL ADDRESS IN TEMPORARY REGISTER TO CURRENT SECTION ID VALUE
EQUAL (CODE IN MEMORY)
230
COMPUTE TARGET ADDRESS FROM IMEM_NR_BASE PLUS OFFSET
NOT EQUAL (CODE NOT IN MEMORY)
224
DETERMINE SIZE OF ROUTINE IDENTIFIED BY TEMPORARY REGISTER, STORE IN SECTION SIZE REGISTER
226
SWAP CURRENT SECTION SIZE INSTRUCTIONS FROM DRAM TO IMEM, STARTING AT SYSBUF_UCODE_START PLUS VIRTUAL ADDRESS IN TEMPORARY REGISTER
228
CURRENT SECTION ID=VIRTUAL ADDRESS IN TEMP REGISTER
A
A

FIG. 4A

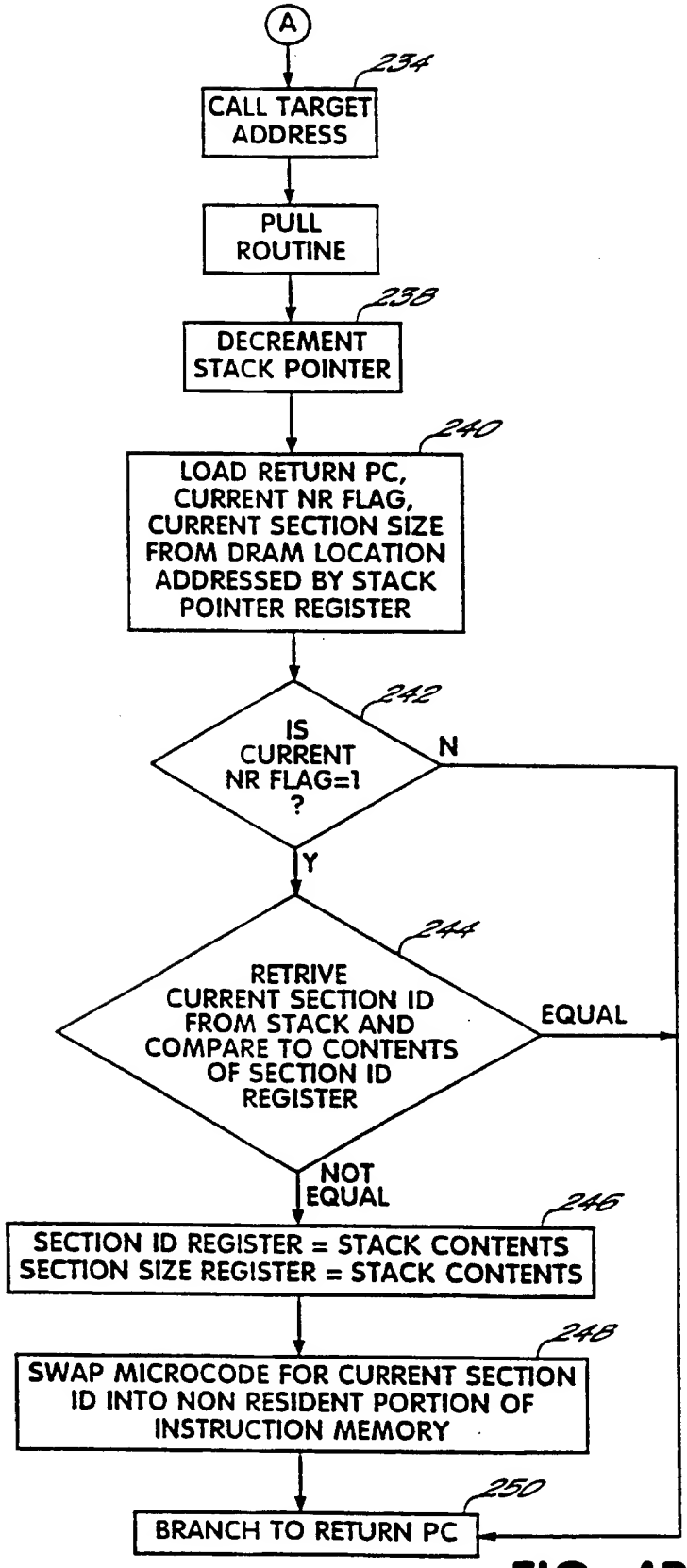


FIG. 4B

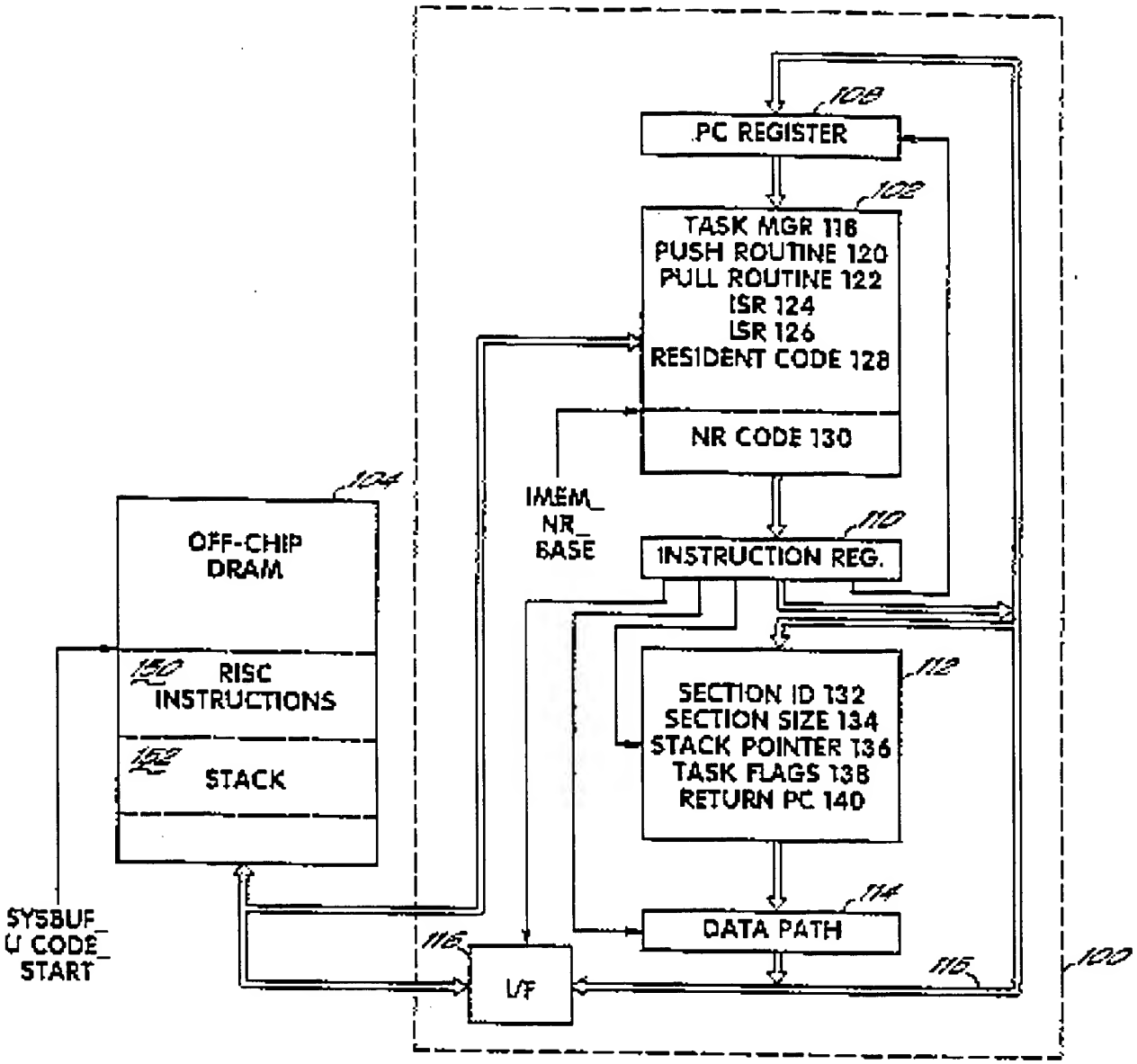


FIG. 1

FIG. 2A

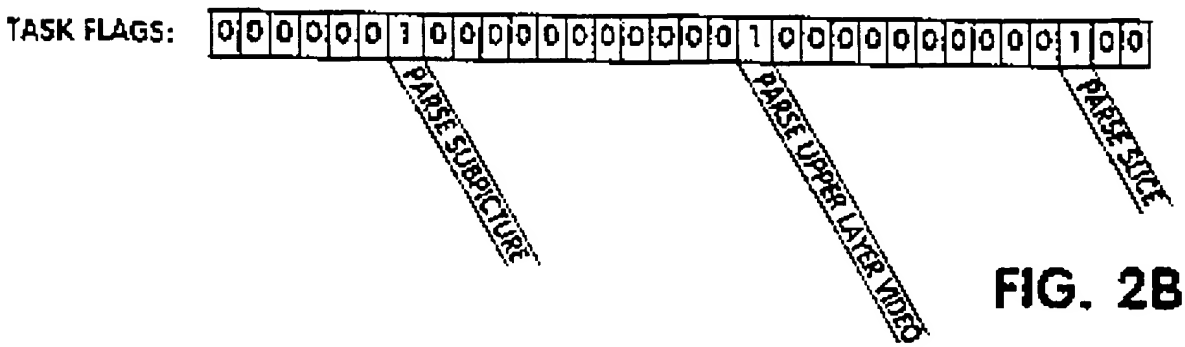


FIG. 2B

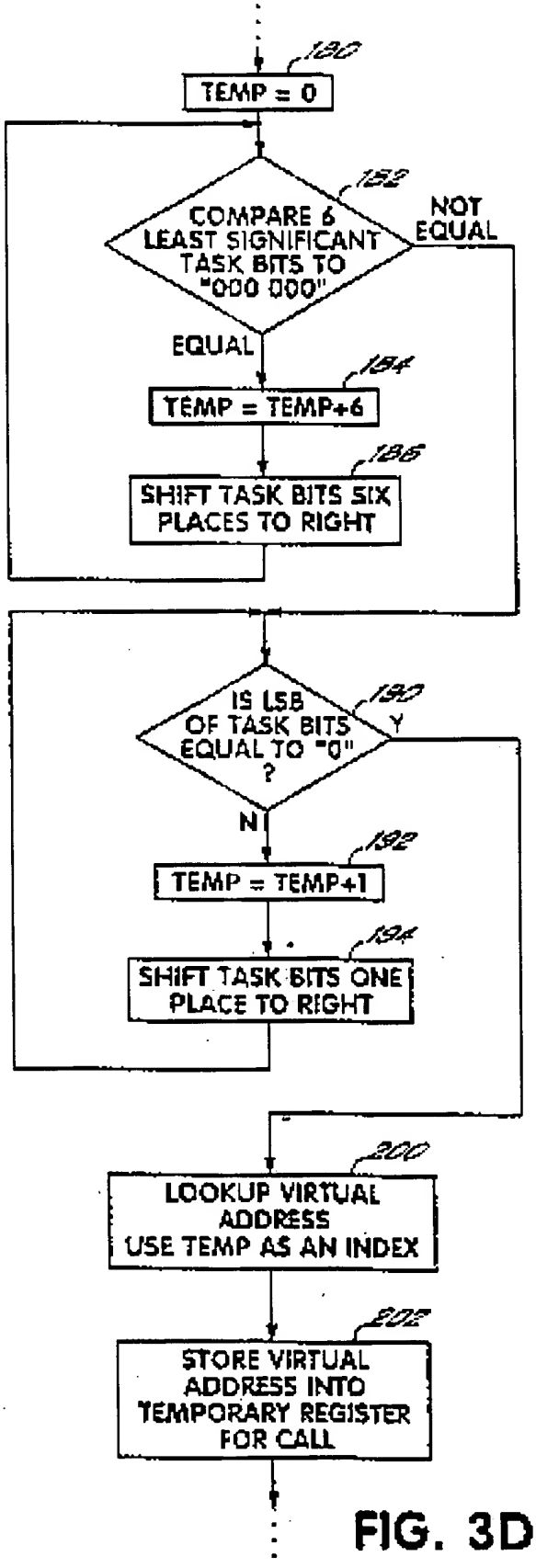


FIG. 3D

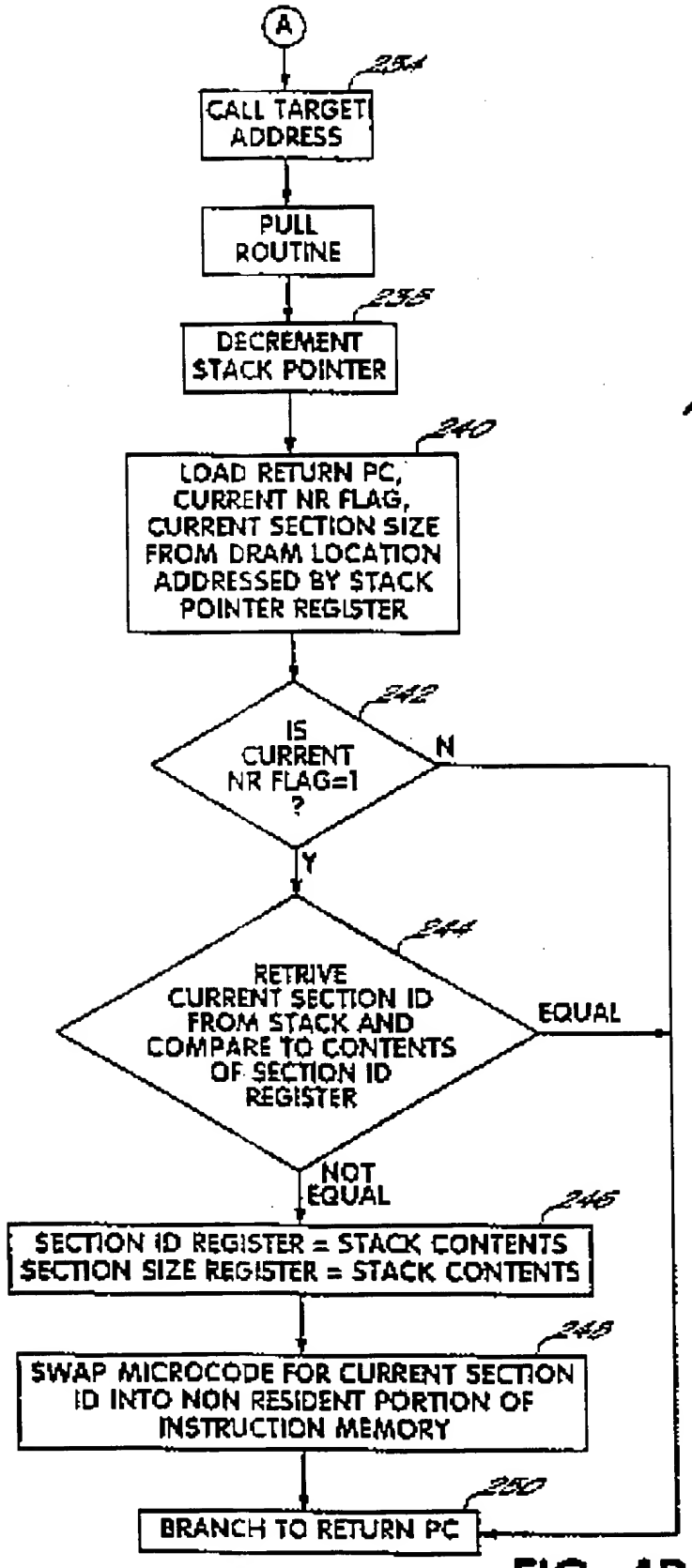

A
234
CALL TARGET ADDRESS
PULL ROUTINE
235
DECREMENT STACK POINTER
122
240
LOAD RETURN PC, CURRENT NR FLAG, CURRENT SECTION SIZE FROM DRAM LOCATION ADDRESSED BY STACK POINTER REGISTER
242
IS CURRENT NR FLAG=1 ?
N
Y
244
RETRIVE CURRENT SECTION ID FROM STACK AND COMPARE TO CONTENTS OF SECTION ID REGISTER
EQUAL
NOT EQUAL
245
SECTION ID REGISTER = STACK CONTENTS
SECTION SIZE REGISTER = STACK CONTENTS
248
SWAP MICROCODE FOR CURRENT SECTION ID INTO NON RESIDENT PORTION OF INSTRUCTION MEMORY
250
BRANCH TO RETURN PC

FIG. 4B

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46, 9/42

A3

(11) International Publication Number: **WO 98/54643**

(43) International Publication Date: 3 December 1998 (03.12.98)

(21) International Application Number: PCT/US98/09138

(22) International Filing Date: 5 May 1998 (05.05.98)

(30) Priority Data:
08/866,419 30 May 1997 (30.05.97) US

(71) Applicant: SONY ELECTRONICS, INC. [US/US]; 1 Sony Drive, Park Ridge, NJ 07656–8003 (US).

(72) Inventors: OZCELIK, Taner; 542 Military Way, Palo Alto, CA 94306 (US). GADRE, Shirish, C.; 1265 N. Capitol Avenue #78, San Jose, CA 94132 (US).

(74) Agents: HUMPHREY, Thomas, W. et al.; Wood, Herron & Evans, L.L.P., 2700 Carew Tower, Cincinnati, OH 45202 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

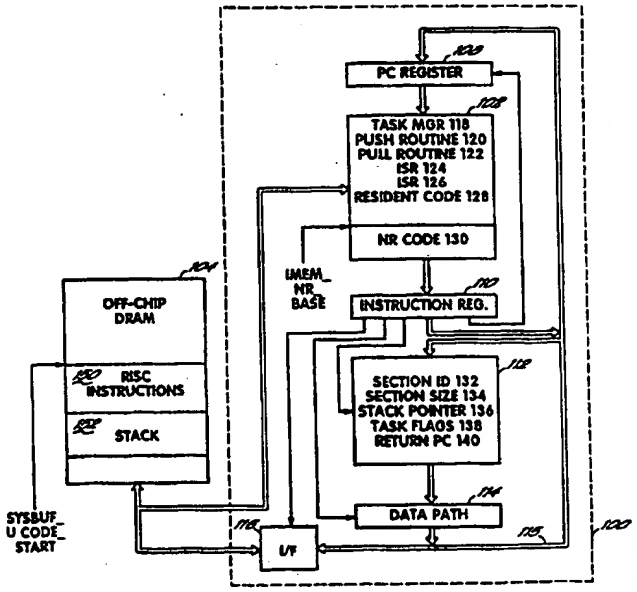
Published
With international search report.

(88) Date of publication of the international search report:
25 February 1999 (25.02.99)

(54) Title: TASK AND STACK MANAGER FOR DIGITAL VIDEO DECODING

(57) Abstract

A reduced instruction set CPU (100) is programmed to provide software–controlled task management, a stack (152), and to manage virtual instruction memory (150). The CPU performs a task management procedure (118) in which the CPU repeatedly checks (168) task flags, and if a task flag is set, performs the task associated with the set task flag. If multiple task flags are set, the highest priority task of those associated with set task flags is performed. Whenever a subroutine call is needed, the subroutine call is implemented by calling (172) a stack management routine. The stack management routine retrieves and stores (216) a return address into a location in DRAM (104) identified by a stack pointer, increments (218) the stack pointer, and then executes (234) a CALL instruction, causing program execution to sequence to the desired subroutine. At the end of each subroutine, a RETURN instruction is executed, in response to which, program execution returns to the stack management routine, and the stack management routine decrements (238) the stack pointer, loads (240) the previously–stored return address from a location in DRAM identified by the stack pointer register, and then causes (250) program execution to sequence to the loaded return address. The stack management routine also provides virtual instruction memory management, by determining (220, 244) whether a routine is resident in the on–chip instruction memory (102) available to the RISC CPU prior to calling or returning to the routine. If not, the virtual instruction memory management routine transfers (226, 248) the desired routine from off–chip DRAM (104) into the on–chip instruction memory, and then executes the call or return.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

INTERNATIONAL SEARCH REPORT

Intr ional Application No
PCT/US 98/09138

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/46 G06F9/42

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	C. BAUDRION ET AL.: "Asynchronous Task Scheduler" IBM TECHNICAL DISCLOSURE BULLETIN, vol. 14, no. 10, March 1972, pages 3192-3193, XP002075013 New York, US	1,5,6
Y		7
A	see the whole document	2-4
	-/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
3 December 1998	23. 12. 98

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Wiltink, J

4

Form PCT/ISA/210 (second sheet) (July 1992)

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	ANZINGER G A: "Managing PA-RISC machines for real-time systems" HEWLETT-PACKARD JOURNAL, vol. 44, no. 4, August 1993, pages 31-37, XP000392979 ISSN 0018-1153, palo alto, ca, usa see page 32, right-hand column, line 29 - page 36, left-hand column, line 13; figures 1-4	7
X	ANONYMOUS: "IOP Task Switching." IBM TECHNICAL DISCLOSURE BULLETIN, vol. 33, no. 5, October 1990, pages 156-158, XP000107413 New York, US	1
A	see the whole document	2-7
A	B.C. BEARDSLEY ET AL.: "Masking Peripheral Interrupts" IBM TECHNICAL DISCLOSURE BULLETIN, vol. 24, no. 11A, April 1982, pages 5362-5363, XP002075014 New York, US see the whole document	1-7
A	HILLS T: "STRUCTURED INTERRUPTS" OPERATING SYSTEMS REVIEW (SIGOPS), vol. 27, no. 1, January 1993, pages 51-68, XP000336450 New York, USA see page 56, line 31 - page 57, line 30	1-7
X	DE MARCHIN P: "Multi-level nesting of subroutines in a one-level microprocessor" COMPUTER DESIGN, USA, vol. 15, no. 2, February 1976, page 118, 120, 122, 124 XP002086079 ISSN 0010-4566	8-10
A	see the whole document	13-16
A	EP 0 565 194 A (PHILIPS PATENTVERWALTUNG ;PHILIPS ELECTRONICS NV (NL)) 13 October 1993 see abstract see column 2, line 7 - last line; claim 1	8-10
A	STEPHEN B. FURBER: "VLSI RISC architecture" 1989 XP002086080 pages 231-233: Branch and Branch with Link Instructions see the whole document	8-10

-/--

4

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category '	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
Y	ANONYMOUS: "Transparent Dynamic Subroutine Loader" IBM TECHNICAL DISCLOSURE BULLETIN, vol. 28, no. 12, May 1986, pages 5535-5537, XP002086668 New York, US	11
A	see the whole document	12-16
Y	EP 0 562 617 A (HITACHI LTD) 29 September 1993	11
A	see abstract see column 2, line 49 - column 3, line 32; claim 1; figures 1,4	12-16
A	EP 0 472 386 A (TEXAS INSTRUMENTS INC) 26 February 1992 see abstract see page 2, line 26 - line 40; claim 1; figure 2	11

INTERNATIONAL SEARCH REPORT

Int. ational application No.
PCT/US 98/09138

Box I Observations where certain claims were found unsearchable (Continuation of item 1 of first sheet)

This International Search Report has not been established in respect of certain claims under Article 17(2)(a) for the following reasons:

1. ☐ Claims Nos.:
because they relate to subject matter not required to be searched by this Authority, namely:

2. ☐ Claims Nos.:
because they relate to parts of the International Application that do not comply with the prescribed requirements to such an extent that no meaningful International Search can be carried out, specifically:

3. ☐ Claims Nos.:
because they are dependent claims and are not drafted in accordance with the second and third sentences of Rule 6.4(a).

Box II Observations where unity of invention is lacking (Continuation of item 2 of first sheet)

This International Searching Authority found multiple inventions in this international application, as follows:

see additional sheet

1. ☒ As all required additional search fees were timely paid by the applicant, this International Search Report covers all searchable claims.

2. ☐ As all searchable claims could be searched without effort justifying an additional fee, this Authority did not invite payment of any additional fee.

3. ☐ As only some of the required additional search fees were timely paid by the applicant, this International Search Report covers only those claims for which fees were paid, specifically claims Nos.:

4. ☐ No required additional search fees were timely paid by the applicant. Consequently, this International Search Report is restricted to the invention first mentioned in the claims; it is covered by claims Nos.:

Remark on Protest

☐ The additional search fees were accompanied by the applicant's protest.

☒ No protest accompanied the payment of additional search fees.

FURTHER INFORMATION CONTINUED FROM PCT/ISA/ 210

This International Searching Authority found multiple (groups of) inventions in this international application, as follows:

1. Claims: 1-7

   Task management in a RISC CPU

2. Claims: 8-10

   Subroutine call processing with stack management in a RISC CPU

3. Claims: 11-20

   Subroutine call processing with virtual instruction memory management in a RISC CPU

INTERNATIONAL SEARCH REPORT

information on patent family members

Inte ional Application No
PCT/US 98/09138

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 0565194 A	13-10-1993	DE 4211966 A	14-10-1993
		JP 6168125 A	14-06-1994
EP 0562617 A	29-09-1993	JP 5274152 A	22-10-1993
		US 5526519 A	11-06-1996
EP 0472386 A	26-02-1992	US 5442764 A	15-08-1995
		DE 69128008 D	27-11-1997
		DE 69128008 T	12-02-1998
		JP 6149760 A	31-05-1994